X-641-73-4
PREPRINT

COMPUTER SOFTWARE DOCUMENTATION

P. A. COMELLA

JANUARY 1973

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

X-641-73-4

COMPUTER SOFTWARE DOCUMENTATION

P. A. Comella
Laboratory for Space Physics
Goddard Space Flight Center
Greenbelt, Maryland

January 1973

CONTENTS

COMPUTER SOFTWARE DOCUMENTATION

P. A. Comella
Laboratory for Space Physics
Goddard Space Flight Center
Greenbelt, Maryland

ABSTRACT

This paper is a tutorial in the documentation of computer software. It presents a methodology for achieving an adequate level of documentation as a natural outgrowth of the total programming effort commencing with the initial problem statement and definition and terminating with the final verification of code. It discusses the content of adequate documentation, the necessity for such documentation and the problems impeding achievement of adequate documentation.

I. INTRODUCTION

The sad state of computer software documentation is a thorn in the side of all associated with the computing field. The literature abounds with advice concerning the content and the format of documentation [1,2,7,8,12,15].

Managerial seminars develop methods to cajole and coerce designers, programmers, coders et al. to document [25,26,27,28]. But the problem remains.

From the midst of the lamentations and hand-wringing over this plight comes a cry - a very loud and very persistent - "Standardize the Format of Documentation". Other voices suggest alternatives [3,5,16,19,20,21].

This paper investigates the area of computer software documentation: the problems, the necessity of solving these problems, the content of adequate documentation, methods of documentation and an evaluation of them. It discusses a methodology for achieving a good level of documentation and the implications of this methodology in the areas of system design, programming, coding, testing and debugging. Lastly, it suggests areas where it is feasible to develop realistic tools which could aid the documenter in his documentation efforts.

II. WHY DOCUMENTATION?

Any manager who has faced the problem of project continuity in the face of employee turnover knows "why documentation".

Any designer modifying an existing system or merging his system with an existing system comprehends "why documentation".

Any programmer modifying an existing program or interfacing his procedure with another's routines understands "why documentation".

And, of course, let us not forget the user, who wishing to use the fully operational, completely debugged, exhaustively tested system, must find out:

1) how to use it; and

2) having used it must ascertain why his data caused an abnormal termination of the system with no output clues as to the cause of termination.

He, assuredly, understands "why documentation".

Good documentation because it is inextricably bound up in each facet of the project from conception and design to coding, testing and acceptance, results in a formalization of the programming effort and this formalization serves as a discipline in creating a programming methodology out of what is, today, still much of a programming art.

Good documentation is an historical record of the implementation of a system. It is a vehicle for communicating the intended functions of the system, the actual functions of the system, and how the system performs these functions. It provides evidence that the system works.

Good documentation also communicates what the system is NOT supposed to do. This is very, very important. After all, the system consists of a finite set of imperatives which can operate on a bounded set of data and it is absolutely essential to know what limitations the system imposes. Thus, the user can know whether the system, operating on his set of data, will output a correct solution. If not, he can discover whether the system is modifiable to his needs and if so what is necessary to modify it.

Good documentation, in its archive function, can also serve as a tutorial in systems design, programming and coding and can result in an improvement of methodologies in these areas.

Good documentation in requiring clear expression of concepts, definitions and functional specifications can prevent distortion of ideas that result in a system's being improperly or suboptimally implemented. It is a tool for project control and evaluations because its production and completion at each phase of a project demonstrates that the phase itself is completed satisfactorily. It is the project's working paper. It makes the system implementation visible and allows for orderly development of subsystems.

Good documentation is also a record of design phase decisions, a record of the alternatives chosen and why as well as a record of the alternatives rejected and why. It is a record of the implications of the decisions: an explanation of the behavior of the system in its operating

environment - a critical inclusion for change of environment frequently results in system malfunction, the result of some unnoticed and undocumented hardware (or software) dependency absent in the new environment.

Ultimately, properly executed documentation frees resources, both human and computing: those consulting good documentation can ascertain the scope of a system and hence evaluate whether a given system is obsolete and should be scrapped or is functional but should be modified or extended. What and how to modify or extend is evident as well as the side-effects of such amendations. Thus good documentation helps to reduce duplication of human effort and unnecessary redundancy in computer systems.

III. THE DIFFICULTY OF ACHIEVING GOOD DOCUMENTATION.

Whenever the documentation effort is not reviewed as an if and only if proposition vis à vis the design, programming and coding effort, the documentation is apt to be inadequate or non-existent. At most it becomes an afterthought of dubious utility, a boring exercise inattentively executed.

Documentation is an area of the computing field where computing personnel demonstrate the least competence. After all, who has learned to document? Programmers are taught to CODE in whatever the programming language! Not much stress is laid on program design and the documentation usually consists of a minimally, internally annotated program that executes a test case of not necessarily critical importance.

Little value is placed on the importance of the design and design decisions while the executing code regardless of its goodness is of premium value. Where is the incentive to document?

Adequate documentation can never occur until the design, programming, coding, etc. are regarded as completed if and only if the documentation is completed.

Lastly, proliferation of hardware devices, programming languages and their accompanying compilers, and software systems, coupled with inadequate assessment of existing documentation procedures, make it difficult to formulate documentation procedures that really work.

IV. THE CONTENT OF DOCUMENTATION

Documentation communicates a message - here a (total) description of a computer software system. Ideally any query concerning the software system will find its answer in the message of the documentation for that system. Of paramount importance is the content of documentation (not necessarily its format). Documentation must communicate effectively concerning the proper operation of the given software system. It must describe the system itself as well as the resources used to develop the system. It must describe the environment of the system and the sub-systems that comprise the system. It must describe the problem that generated the idea for the system, i.,e. the purpose of the system. It must describe the input set, i.e. the solutions, and the algorithms

and procedures which transform an input subset into the appropriate output subset.

Efficient documentation has as its offspring the generation of new concepts and the revision of old methodologies in problem solving because the system development has become visible, available and complete and hence evaluable.

There is broad general agreement concerning the content of documentation. Basically adequate description of a computer software system involves:

1) documentation for management functions
2) documentation for user functions
3) documentation for operational functions
4) documentation for analysis, design and programming functions

Documentation for management is a justification of the system. It outlines the problems and needs prompting the proposal of the new system (or existing system modification) and the benefits which will accrue because of its implementation. It is a statement of the broad conceptual design of the proposed system with particular emphasis on its being a good solution to the problem in comparison with other alternatives. It provides facts in the realm of dollars and cents, personnel requirements, time schedules, computer resource requirements necessary for measurement and evaluation of the system and for sound management decision-making. The format of the presentation must be satisfactory to management and is not

the subject of this paper. For discussion of format and other specifics of management requirements, see references [7,12,25,26,27,28].

Documentation for user functions consists of a general system description with appropriate definition of terms enabling the user to ascertain the functions of the system and its limitations, the flow of the system, the domain of the input, the range of the output, the algorithms and procedures that transform the input into output, the procedures for preparing input, the error handling procedures for detection of bad input. It contains instructions for preparing the input and illustrative test cases. Gray and London [12] discuss user documentation in adequate detail although from the point of view of standardizing documentation.

Documentation for operational functions describes the environment in which the system must operate; the hardware devices required, their configuration and set-up. It tells how to start up the system as well as how to restart in case of failure. It identifies the I/O devices and files and contains a detailed description of the data preparation procedures. It states storage requirements both main and peripheral and timing requirements: CPU and I/O in meaningful units(perhaps defined within the body of the documentation). Again Gray and London [12] is a good reference for content of operational documentation.

Documentation for analysis design and programming functions is the category of documentation which is the subject of this paper.

Analytical documentation is a detailed statement of the problem and design. It defines the problem completely according to its input, output and the functional specifications - the sequence of logical states transforming the input to output. It defines the operating environment of the system - the computer and peripheral devices, the operating system, the command and control language, and the programming language(s) of the implementation. It defines (and orders with respect to implementation) the sub-systems comprising the system with their appropriate task generations; the structure of the data base containing the input files to the system as well as the output files. This includes the assignment of files to specific hardware devices, data set name by which the file is known to the system, organization of the file and format of records within files and the relation of contents to input/output or intermediate processing. It discusses file maintenance in terms of the update and retrieval mechanisms. It creates the testing systems and sequences stating critical test points and paths and acceptance criteria. In the functional specifications it notes explicitly where errors can be detected and how such errors are to be handled.

The analytical documentation imposes structure on the problem definition, translating the initial statement into a meta-language from which restatement the problem can be translated (directly) into the selected programming language(s).

It specifies, as noted above, the order of tasking and sub-tasking to occur in the system implementation and the order of programming the functional specifications, thus, indicating the general logic flow and control of the system . It specifies the structure of the data base and defines the files comprising the data base. It describes the content and format of the files. It specifies the interfaces with the operating system and explicitly states what the system can and cannot do. In this design phase, too, the testing specifications are written.

From the analytical specifications the programs are designed. Programming documentation describes the algorithms and procedures of the functional specifications, the detailed logic of each procedure which transforms the procedural input to procedural output. It specifies the interfaces with its sub-systems. (All of this should occur in a meta-language of the programming language chosen. For example, if the selected programming language is ALGOL the logic should be written in ALGOL-like constructs). It defines the variables and functions used in the computations. From these programming specifications the coding can directly proceed. The coded program forms an integral part of the documentation. It is especially valuable where cross-referencing with programming documentation is present. (Appropriate tools for clarification in this phase of the documentation might include glossaries and indexes to provide definitions and cross-referencing, schematics and figures to illustrate logic flows and the

structure and contents of the data base).

This explicit statement of the analysis, system design, programming and coding is the documentation.

V. THE QUESTION OF STANDARDIZATION

The question of standardization is implicit in any discussion of documentation. As suggested in the introduction management solutions to the documentation problem frequently lie in the realm of standardization of the format of documentation.

Part of the rationale behind this leaning towards standardized formats is the idea that:

1) anyone can fill out a form with proper guidance; thus;

2) the documentation problem becomes a managerial problem with attendant solution lying in the comprehensible (to management) areas of forms design, forms distribution and on-the-job training.

There is another reason, however, more obscure but more insidious in its pervasive influence on the computing industry - an under estimation of the complexity of the mechanisms and methodologies of problem specification and systems design and implementation. Dijkstra has stated this quite eloquently [9]. Thus, while management's aim is to achieve a system of documentation that is easy to prepare (thinking that is why people don't document), comprehensive in its description of the problem, solution and use of the system, it sabotages its goal to a large extent, not by its insistence on standardization per se bt by its selection of format as the criterion for standardization.

Standardization imposes a discipline on the user of the standardized procedure and so creates a focal point in the approach to a problem. Thus, in this case management has made the format and not (necessarily) the content of documentation, as originally intended, the focus of the documentation effort. But the critical need is how to create well-designed and implemented systems that are adequately documented: to do away with the transformation of input to output via alchemy with the functional specifications tucked away in the privacy of the author's mind. This writer is convinced that people don't document because they, recognizing that documentation is part of software development inseparable from the analytical, design, programming, coding and testing phases, don't know how to achieve this integration of documentation with the problem definition and solution. And to resolve this frustrating dilemma they skirt the documentation issue and thus diminish their powers of creativity in system design and implementation: this lack of graphic statements leads to imprecise sub-optimal systems containing (undocumented) side-effects.

So it turns out that management has pinpointed the solution - standardization - but related it to the wrong problem - the format!

But if the solution of standardization is applied to the problem of content it becomes apparent that the proper thrust

should be towards developing a methodology of problem definition, systems design and implementation out of which flows naturally the documentation.

The next section is a discussion of such a standard - the methodology of top-down definition, design, programming, coding and testing. The methodology has implications in the critical areas of demonstration of program correctness, test and validation of systems, and debugging of code. Dijkstra [9], Mills [19,20], and Parnas [21,22] have excellent articles describing implementations employing this methodology.

The immediate consequences, in relation to the topic of software documentation, of this methodology are as follows:

1) the problem is well-defined and documented
2) the design, programming and coding are well-considered, approach optimal and are documented
3) the system is usable, amendable and extendable and the know-how is in the documentation
4) critical parts within the program are noted and the testing documentation explicitly demonstrates the operation of the code along these paths. Demonstration of program correctness becomes feasible.

Thus the system is well-executed and the documentation is well-executed. Furthermore, and very importantly, the content of the documentation is a natural outgrowth of the system implementation.

VI. A METHODOLOGY FOR SOFTWARE DOCUMENTATION

This section describes a highly structured top-down approach to the problem of generating a software system and its attendant documentation. It is analogous to the construction of a tree with the root of the tree the problem statement; the first level the embedding of the problem in its operating environment; the second level the specification of internal system invariants and the interfacing of these invariants with the external environment; the third and deeper levels the representation of the functional specifications and sub-specifications in such a way that those specifications in closest propinquity to the root exert the greatest influence in constraining the system to meet problem objectives. Furthermore, the outer level nodes always determine the interfaces with nodes at the next most inner level. Influences and constraints percolate downward, never upward or laterally.

The aim will be to establish an equivalence (in meaning) between the system and its documentation and to allow the documentation to keep abreast of system development.

The raison d'être for a top-down approach to systems design and programming lies in the fact - as Hoare [14] so aptly states: ". . . all of the properties of a program and all of the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning,"

(p. 576). Armed with this knowledge it makes sense to derive a methodology that makes it possible to approximate this realization.

Dijkstra and Mills [9,19,20] among others advocate the application of the concepts of structured programming - a method of programming requiring all functional specifications of a procedure to be expressed using only the following forms of statements:

1) sequence
2) if . . . then . . . else
3) do . . . while

with all functional specifications having but one set of input and one entry point and but one set of output and one exit point. (This advocacy of go-to-less programming has its origins in the theorems of Böhn and Jacopini [4]. From the point of view of documentation, verification and testing of systems such as approach is highly desirable because given a program written without go-to's much can be said about whether the program does what it is supposed to do.

However, because a go-to-less language may not be available to the reader of this paper, the subject of structured programming will not be pursued further here, but instead the paper will describe a top-down methodology embodying the spirit of structured programming. (As an aside, Mills claims such aims can be achieved even with the 'go to' allowed [20]).

The discussion will suggest a methodology for problem definition, system design and specification, including testing procedures, program design and coding, all with a view towards producing an optimal system optimally documented.

A. Problem Definition

The first step is to wrest the problem statement from the requestor in order to arrive at a problem definition. The guiding principle must be the realization that there is almost always dichotomy between what the user really wants or can have and what he thinks he wants - this is particularly true of the naive user of computer systems.

Thus, the designer (consultant) in eliciting the request must also elicit the purpose of the request. This will aid him in pursuing a line of questioning, the answers to which will culminate in a workable problem definition. The user, while comprehending that total implementation will frequently occur through time, i.e. the implementation will occur in stages - frequently operates under the premise that problem definition should likewise take place across time. This is an erroneous viewpoint having as end result - even under an assumption of correct problem definition - which is unlikely - suboptimal design, patched programs and code, introduction of undesirable side-effects and excessive debug time! The designer must assist the requestor in achieving a correct, and complete statement of the problem before proceeding with design and analysis.

Secondly, the designer must ascertain what are actually system parameters and what are system invariants. Here, particularly in a modeling situation the "solution" is frequently only a prelude to the solution and the "constants" but zeroth order estimates of the solution constants (this may motivate a design embedding no constants within the body of the functional code). Frequently, too, entire sections of code will be replaced: the functions themselves are parameters and will undergo tremendous revision. Such considerations, if known beforehand, can influence the design hugely, while lack of knowledge of this consideration can render the procedures difficult to modify. Forewarning, too, might influence the designer to choose a high level programming language for which an optimizing compiler exists so that functions can be readily coded and modified.

Another area for scrutiny is the output specifications. The user requests what he knows about and what he thinks expeditious. Underlying a request for tabular output may be a need for plotting because the user is unknowledgeable concerning capabilities for computer plotting or because he thinks it will take too long to include the plotting right off. It is best to elicit these needs before the programming design and coding commence, for even if the initial implementation does not execute these options a place in the code can be set aside for later insertion of code using dummy procedures, etc. This is superior to patching a running

program: patching frequently introduces side-effects, increases the difficulty of testing and debugging and decreases code readability. It is necessary to note here that testing is a design function: the testing specifications flow from the design considerations and the problem specifications.

Input considerations are of crucial importance, too. It is necessary to know the answers to these questions before proceeding with system design:

What is the input?

Where do the data come from?

What does the input look like?

Is its format inflexible? This is very important. Many times the fixed formatted data is actually variable and the coding based on the fixed format premise, while efficient, has constrained the design too strongly percolating its influence through many levels of coding so that changes in format require - sometimes - major code revisions and that inevitable patching. The designer must be aware of this problem and must make the requestor cognizant of the implications of his specifications. Thus, the designer must make certain that the requestor can really live with his specifications.

On the other hand, there are certain data sets whose formats are invariant (tapes from a satellite, for instance) but which could contain errors. Thus, the problem of error detection and error handling and correction requires attention.

What is the probability of error occurrence?

Of what type?

What is to be done when the error occurs - abort, ignore, correct?

What are the indications that an error has occurred?

The relative sophistication of the expected user is also a consideration in the design of the input subsystem.

On output design decisions are made concerning output media, optionality of output data sets, format of solution data sets, content, format and location (on-line or off-line with respect to the solution data sets) of error messages of adequate content to locate the source of error - a challenging problem.

The requestor must understand the importance of the problem statement - the completeness of the statement ultimately determines the implemented system's usability.

The last phase of these pre-design consultations is the presentation of the request - as the designer perceives it - to the requestor (preferably for his signature) - this signals concurrence with the problem statement and enables the designer to proceed to the analysis and design of the system.

Addenda and amendations to the proposal should be stated in writing - problems can undergo striking metamorphoses in the course of development and to maintain clarity of problem definition and requestor - designer accord such a policy is wise.

This phase completed the problem at that given instant of time is defined. This definition can be achieved even in a research environment - a point that is frequently argued as not possible - because the problem statement clearly indicates the variables and constants of the system . Even though it is not explicitly known when or how parameters will change in the course of development it is known that they may and the design can allow for this. Thus, from the design point of view the problem is defined.

B. The System Design

With problem statement in hand the designer commences the analysis and specification of the system - the problem solution.

In the top-down approach, given the problem statement and a definition of the available computing resources, the designer proceeds to define the problem as a system embedded in an environment composed of a subset of these resources: hardware devices, operating systems, compilers, assemblers, etc. Criteria for selection of each resource are explicitly stated as well as the implications, i.e. constraints imposed by the selection. Reasons for rejection of alternatives - where available - are also explicitly stated.

This environment with its attributes exerts external influences upon the developing system, independent of the constraints which the problem statement imposes. But it is critical to note: the external environment acts first and the system must conform to these external behavior demands

before it can respond to the internal demands of the problem itself. Consequently, in the top-down methodology the control commands that correctly interface the embryonic system with its outside world are written first. This places the problem definition in a proper perspective; but not only that, it makes the system in skeleton form known to the computing system.

And, much to the joy of all concerned, satisfies that magical need to get running - but in a very special way - a hierarchical way such that encompassing code is always executing and "fully" tested before the next lower level of executable code is created and integrated into the system. "Fully" means at least to the point of determining the syntactical validity of constructs.

Now the system communicates with this environment not only through this command and control language but also through its I/O requests. Furthermore, it is usually in terms of I/O that the user has stated his most stringent, least flexible requirements: in terms of satisfaction of user requirements the I/O area can be most critical.

Thus, it is good top-down philosophy to define these communication paths at the next level; in IBMese the medium would be the Job Control Language (JCL). The JCL specifications, however, are a function of the data base design. Hence, creation of JCL controlling user I/O requests implies prior definition of the data base characteristics: the files comprising the data base; the format (organization) of each

of these files, the data set names referencing these files; the criteria stipulating the organization and the implications of the type of organization selected. (Schemata and tables serve useful purposes in illustrating the structure of data bases and organization of files).

This JCL funnels user input through the system environment connecting it with its processing program for functional transformation into output conforming with user requirements and transfers these results to the appropriate output media.

Thus, this data base and its JCL connect user to system and force compliance with the user's most stringent requirements at the outermost levels.

Next comes the creation of the control and functional code in order of dominance commencing with the coding of the critical nucleus (that section of code which controls the primary specification of functions) and tasks, its testing and integration into the system as follows:

1) Nucleus of control code for the i+1st level is created at the i^{th} level; functional code created at the i+1st level has its interfaces defined at the i^{th} level and only at the i^{th} level;

2) The i+1st level of code is tested and integrated with the system (which exists, is executing and has been implemented and tested through the i^{th} level of definition).

Steps 1 and 2 are iterated until the system is fully implemented.

It is important to note several features of this top-down methodology:

1) The programmer is able to carry out his design structure in code

2) The programmer is able to design a testable system

3) The documentation is a natural outflow of the design and test functions. The following paragraphs amplify these points.

Sub-task specification and functional specifications are always carried out in executable code; for example, the appearance of a CALL SUBR (ARGI, . . ., ARGN) in the i^{th} level of code signals (and is the only signal) the design need to create at the i+1st level of code a procedure, known to the system by the name SUBR which operates on the input, (an already defined subset of the arguments, ARGI,. . .) to produce the output, (defined within SUBR as a subset of ARGI, . . .). Further, the function (sub-task) is always referenced through its interface defined at the supervisory level; (ideally) each function (sub-task) reports to one supervisor only (to minimize connectivity see below).

Observe: only that information which enables the function to be correctly coded without introduction of upward or lateral side-effects is supplied in the interface, nothing more and nothing less. The aim here is to minimize program connectivity - connections are assumptions that modules make about one another - and to prevent, in modification of a module, violations of assumptions other modules make about the module being changed.

(Here, to modify one simply proceeds down the code-tree to the affected node(s) and replaces the function or task by another which satisfies the same criterion of correctness).

Sub-task specification permits maintenance of program integrity and permits demonstration of program correctness - the documentation of the specifications; while the bottom-up approach of supra and lateral task specifications pays inadequate attention to the testing problem - the uncovering of errors, the debugging problem, the locating of the roots of the error and their subsequent correction, and the connectivity problem, all of which have repercussions in demonstration of program correctness and hence documentation.

Aspects of testing must enter into design-stage decision making so that the resulting system has the following characteristics:

1) The structure of the program forms the basis for design of the test

2) the number of relevant states (states to be tested) is of reasonable extent

3) the relevant states are indeed testable.

Test planning proceeds as follows:

1) determination of the extent of testing

2) identification of testable states

3) ranking of testable states in order of importance according to criteria derived from critical properties the problem solution must possess in order to satisfy user requirements

4) selection of relevant states using the ordering of step 3

5) structuring of program design so that these relevant states are testable

6) development of a set of test data which forces the system into all of its relevant states

7) verification of code by execution of the program using the test cases. Note: the statement of properties which the system must have to satisfy user requirements influences the test plan most strongly. This statement is a series of assertions describing the behavior of the program, that is, the effects of computation on the input set.

Such assertions occur:

1) at the end of program/procedure stating what the correctness of the program/procedure means

2) at the start of the program/procedure concerning satisfaction of initial conditions

3) at some point along each loop

4) at points of functional specifications

The test plan incorporates these assertions into the program. Execution of the test cases demonstrates program correctness as follows: whenever the initial conditions are true and the assertion is true at each critical point along the path then the final assertion is true. Conversely, if the program fails at some point, the path output can locate the source of the error, thus expediting correction.

Note: The test plan code is permanent code optionally executed - it is an inherent part of the documentation indicating relevant states, critical paths and assertions. Test plan code is necessarily executed during initial system creation and subsequent modification phases.

The test plan discussed in the preceding paragraphs is an integral part not only of the program structure but also of the documentation: execution of the test cases:

1) demonstrates that the program does what it is supposed to do

2) facilitates system modification - in fact, renders it feasible

3) demonstrates what transformations algorithms and procedures effect upon their set of input data

- all primary functions of documentation.

Lastly, in the top-down methodology the program itself becomes a high level flowchart. The outermost levels of the code synopsize the program. Functional specifications are known at their point of origin in the parent nodes by their symbolic source language names. Also, their interfaces are explicitly stated in these references.

Finally, each "box" of this flowchart - node of the code-tree - with its combination of control code defining functional specifications at the next inner level and functional code operating on input data designated at the immediately outer level is a natural unit of documentation. For greater readability Mills [20] advocates limitation of unit size to one page of a computer program listing.

In conclusion, the top-down methodology structures the entire development of the problem solution, standardizing the approach to the analysis, design, programming and coding functions with the outcome of well designed systems, properly coded and tested and adequately documented.

VII. DEVELOPING DOCUMENTATION TOOLS

This section briefly suggests two areas of attention which might offer some opportunities for amelioration of the documentation problem.

The classroom occupies an inadequate place in the documentation effort, its potential as a maker of documentation tools - the programmer, himself, largely ignored.

For the most-part programming courses are coding courses. Instructors almost invariably, stress the coding aspects of projects, indicating to students that cute working code is of inestimable assistance in earning the coveted high grade. Daily admonitions concerning the limited time remaining to get running and debugging drive the student to frenzies of coding and debugging. The design, analysis and documentation aspects of programming as well as their interrelationships receive scant attention. The result - a code first, document later mentality permeates the computing field!

The solution - the development of curricula for programming courses which place coding in its proper relationship to analysis, design, programming, testing and documentation - and, the use if must be, of that infamous lever, the grade, to

underscore the necessity of adequate analysis, design, testing and documentation in the total effort.

Another potential tool is the use of syntax-directed compilers such as Wirth has developed for PL/I (see Mills [19]). This compiler constructs questions from the program being compiled for the programmer's later answering on an interactive system. This effort provides a standard way to elicit programmer response to specific questions regarding his program, the questions themselves, the result of compiler analysis of the program.

VIII. CONCLUSION

Only those programming methodologies which integrate the analysis, design, programming, testing, coding and documentation efforts can have as output well designed systems, adequately documented. This paper describes such a methodology.

BIBLIOGRAPHY

1. ADP Documentation Handbook, Social and Rehabilitation Service, PB-198085, September 1971.

2. A Programmer's Guide to the Goddard Space Flight Center Computer Program Library, X-540-72-114, NASA, GSFC, February 1972.

3. Baker, F. T., Chief programmer team management of production programming, IBM System Journal 11, No. 1, 56-73 (January 1972).

4. Böhm, Corrado and Giuseppe Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, Communications of the ACM 9, No. 3, 366-371 (May 1966).

5. Chief Programmer Teams: Principles and Procedures, Report No. FSC 71-5108. International Business Machines Corporation, Federal Systems Division, Gaithersburg, Maryland (June 1971).

6. Collins, B. D. and N. B. Acker, Documentation and debugging, Data Management 8, No. 9, 107-115 (Sept. 1970).

7. Computer Program Documentation Guideline, NHB 2411.1, National Aeronautics and Space Administration, Washington, D. C., July 1971.

8. Computer Program Systems for ADP Management: Documentation Standards, X-502-70-157, NASA, GSFC, December 1969.

9. Dijkstra, E. W., The structure of "THE" multiprogramming system, Communications of the ACM 11, No. 5, 341-346 (May 1968).

10. Elmendorf, W. R., Disciplined software testing, Debugging Techniques in Large Systems, 137-139, Prentice-Hall, Inc. Englewood Cliffs, N.J., 1971.

11. Floyd, R. W., Assigning meanings to programs, Proceedings American Mathematical Society Symposium in Applied Mathematics 19, 19-31, (1967).

12. Gray, Max and Keith London, Documentation Standards, Brandon/Systems Press, Inc. Princeton, N.J., 1969.

13. Hill, P. B., The control of large scale software projects, IAG Journal 3, No. 12, 394-405 (December 1970).

14. Hoare, C. A. R., An axiomatic basis for computer programming, Communications of the ACM 12, No. 10, 576-580, 583 (October 1969).

15. Howarth, R. J. and A. L. Lin, An approach to program documentation, Computer Bulletin 13, No. 8, 291-295 (August 1969).

16. J. Katzonelson, Documentation and the management of a software project - a Case Study, Software-Practice and Experience 1, April-June 1971.

17. King, J., A verifying compiler, Debugging Techniques in Large Systems, 17-40, Prentice-Hall, Inc. Englewood Cliffs, N.J., 1971.

18. King, P.J.H., System analysis documentation: computer aided data dictionary definition, Computer Journal 12, No. 1, 6-9 (February 1969).

19. Mills, H. D., Syntax directed documentation for PL360, Communications of the ACM 13, No. 4, 216-222 (April 1970).

20. Mills, H. D., Top-down programming in large systems, Debugging Techniques in Large Systems, 41-55, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

21. Parnas, David L., Information Distribution Aspects of Design Methodology, AD-719863, February 1971.

22. Parnas, David L. and John A. Darringer, SODAS and a methodology for system design, AFIPS Conference Proceedings 31, 449-474(1967).

23. Perry, James W. and William Goffman, Mathematical Formulation of Basic Procedures in Documentation, AD-429098, April 1960.

24. Schlender, Paul, Application of disciplined software testing, Debugging Techniques in Large Systems, 141-142, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

25. Schwartz, M. Herbert and Bruce Beardsley, Systems proposal documentation for senior management, Data Management 8, No. 9, 88-93 (September 1970).

26. Senensieb, N. Louis, Principles of systems analysis and design, Data Management 8, No. 9, 19-23 (September 1970).

27. Verstandig, Harry, B., A Modular Approach to EDP Documentation, Journal of Data Management 7, No. 7, 18-23 (July 1969).

28. Wofsey, M. M., Systems development, conversion and operating plans, Data Management 8, No. 9, 64-65 (September 1970).